①

WL-TM-92-110

DECOMPOSITION STRATEGY FOR THE FUNCTION LEARNING AND SYNTHESIS HOTBED

Timothy D. Ross
WL/AART-2
WPAFB OH 45433-6543

10Q8

System Concepts Group
Applications Branch
Mission Avionics Division

Approved for public release:
Distribution is Unlimited

Avionics Directorate
Wright Laboratory
Wright-Patterson AFB OH 45433-6543

25 Aug 92	Interim 1 Jul 92 - 25 Aug 92

Decomposition Strategy for the Function Learning and Synthesis Hotbed

WU 0100AA13
PE 61101F
PR 0100
TA AA
WU 13

Timothy D. Ross

Avionics Directorate, WL, AFMC
Wl/AART-2
Wright-Patterson AFB OH 45433-6543

Timothy D. Ross, 53215

WL-TM-92-110

This memo describes the function decomposition strategy that is implemented in the Function Learning and Synthesis Hotbed (FLASH). FLASH is a general purpose tool for research in pattern theory.

Function Decomposition, Computational Complexity, Pattern Theroy, Pattern Recognition, and Circuit Optimization.

9

UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

[illegible] INSTRUCTIONS FOR COMPLETING SF 298

[illegible] reports [illegible]

[illegible]

FOREWORD

This technical memorandum was prepared by Timothy D. Ross. This memo describes the decomposition strategy that is implemented in the Function Learning and Systhesis Hotbed (FLASH). This work was carried out in the System Concepts Group, Applications Branch, Mission Avionics Division, Avionics Directorate, Wright Laboratory, Wright-Patterson AFB OH 45433-6543. This study was performed under unit 0100AA13, Pattern Theory 2.

The author wishes to thank Prof. Michael Breen of Tennessee Technical University and Mr. Michael Noviskey of WL/AART-2, Wright-Patterson AFB OH for their many suggested improvements to this memo.

This Technical Memorandum has been reviewed and approved.

Timothy D. Ross

TIMOTHY D. ROSS
Electronics Engineer
System Concepts Group

[illegible]

JOHN E. JACOBS
Chief, System Concepts Group
Applications Branch

Decomposition Strategy for the Function Learning and Synthesis Hotbed

August 25, 1992

1 Introduction

This memo describes the function decomposition strategy that is implemented in the Function Learning and Synthesis Hotbed (FLASH). FLASH is a general purpose tool being developed under the Pattern Theory (PT) 2 project. This memo assumes the reader is familiar with the terminology and results of the PT 1 project [3].

2 The Basic Structure

The basic structure of the decomposition function is shown in Figure 1 using a C based psuedo-code. The objective of *find_partition* is to find the partition of variables whose children have a minimal total DFC. The *construct_children* function simply generates the simple decomposition of f with respect to the partition found by *find_partition*. The children are then decomposed with a recursive call to *decomp*. The complexity of *decomp* therefore boils down to the complexity of *find_partition*.

There are (at least) two reasons why *find_partition* is computationally expensive.

1. There are a lot of partitions (3^n); therefore, it is generally not practical to consider all of them in a search.

```
decomp(f){
        find_partition();
        construct_children();
        for (i=0; i < no_of_children; i++)
                decomp(i-th child);
};
```

Figure 1: The decomp function.

2. Finding the exact *DFC* of the children is difficult, partly because it involves *find_partition*, but also because it must involve (as far as we know) the computation of column multiplicity. Computing column multiplicity will require about as many compares as the cardinality of the function. For total functions, finding column multiplicity has exponential complexity.

Therefore, we must introduce heuristics that reduce these two factors to something sub-exponential. That is, the heuristics must:

1. help us select good partitions for evaluation

2. help us evaluate a partition without computing the exact DFC of the children.

The next two sections address some of the possible heuristics in these two areas.

3 Partition Selection Strategies

Most searches can be characterized by the order of the search and the stopping criteria. The order might be random, be a progression through neighbors, have a specific number of row/column variables, or exclude shared-variables. Data from a partition-survey of the palindrome function suggests that you try similar partitions to ones with low column multiplicity and very dissimilar partitions to ones with high column multiplicity.

Once we have selected a particular order for a search, we must decide when to stop the search. This decision might be based on exhausting all possible partitions or some resource allotment, such as a specified number of partitions or a specified length of time. There are also bounds on DFC based on the number of non-vacuous variables, number of minority elements, and number of cares that could be used in the decision to stop. For example, when you have found a decomposition with a DFC close to a lower bound, there is little payoff in continuing.

There may also be constructive approaches to generating partitions [2, 4]. It has been suggested that grouping variables according to their correlation with the output. It may also be possible to place function values in a partition matrix to minimize column multiplicity and end up defining a partition.

Each of these approaches have a number of parameters that need to be specified and they could also be combined.

4 Partition Evaluation Strategies

The baseline here is to compute the exact DFC of each child. Since this is intractable, we have various figures-of-merit (FOM) that basically approximate the children's DFC. These FOM's include column multiplicity, the number of features, DFC of a simple decomposition, estimated average column multiplicity for each child, and the DFC of a simple decomposition of each child.

The computation of column multiplicity is central to many of the FOMs. Breen [1] poses the general problem of computing column multiplicity along with a number of important properties. Other ideas include stopping the count of column multiplicity when it exceeds a threshold and comparing word-length numbers of bits at a time. Thresholds for column multiplicity are also discussed in [3, Section 5.2.4].

5 The Decomposition Plan

Our intent in developing FLASH is that it be easily modified to explore the above issues about the efficiency and effectiveness of various decomposition methods. Therefore, rather than changing code, we want to select different methods as part of the run-time set-up. A class called *decomp_plan* has been defined to represent the definition of the decomposition method to be used. Within *decomp_plan* there are provisions for specifying the following characteristics of a decomposition strategy.

A decomposition plan includes the following parts.

- an indication of whether or not partition sets are used
- the number of iterations
- a partition selection plan
- a partition evaluation plan
- a decomposition plan for the children of the best partition.

The partition selection plan includes

- a partition selection method, including options for:
  - middle out, based on row/column ratio
  - max to min, based on row/column ratio
  - min to max, based on row/column ratio
  - random
  - construct
  - based on dist from initial part.
- a criteria for stopping partition selections, including options for:
  - exhaustive
  - stop after so many partitions

  - stop after so much run-time
  - stop when some FOM exceeds a threshold.

- a parameter (if required) for the stopping criteria.

The partition evaluation plan includes:

- the number of levels in the evaluation
- an evaluation method for each level.

Finally, an evaluation method specifies:

- the method for estimating DFC, including:
  - function cardinality
  - average column multiplicity based
  - column and row multiplicity based
  - DFC as computed by a specified decomp plan.
- the percentage of f's cardinality that the DFC must be less than to survive the cutoff.
- the decomp plan for the children when estimating DFC.

This decomp plan is a summary of the partition selection and evaluation strategies that will be allowed in FLASH.

6 General form of *find_partition*

We would expect the more accurate evaluations of a partition to be more computationally expensive. This suggests an iterative approach to *find_partition*. That is, we first narrow the set of partitions using a rough (but fast) figure-of-merit and then search this smaller set with a more accurate (but slower) figure-of-merit. This could be generalized into any number of iterations. For each iteration then, we do not just want the best partition considered, we want a set of partitions. The decision becomes whether or not to include a particular partition under consideration, based on its FOM, in the set. Two methods for building this set come to mind.

1. include partitions where the FOM exceeds some threshold
2. for some constant K, keep the best K partitions of those considered

```
for (int i = 1; i < no_of_iterations; i++){
      while ( not_done(i) ) { p = choose_partition( p_set[i-1] ); p_set[i]
                        -> update(p); }; p_set[i] -> reorganize(i); };
```

Figure 2: The find partition function.

It would be possible to combine these two methods. There is also an opportunity to analyze the set of high FOM partitions and adjust the set. An adjustment suggested by the palindrome partition-survey data is to AND or OR partitions from the set together and then add them to the set. The stopping conditions that might be tried now include the ones above, plus having the number of partitions in the set exceed a threshold.

These search considerations suggest the following forms for *find_partition*. The first form, shown in Figure 2, is the most direct.

p_set[*i*] is the set of partitions that resulted from the i^{th} iteration. *p_set*[0] is the set of all possible partitions.

choose_partition would select, perhaps as a function of the iteration number, a partition from its argument *p_set*. When there is no argument, as in the 0^{th} iteration, all possible partitions are considered.

not_done, again possibly as a function of the iteration number, would decide if the set of partitions was up to snuff, i.e. have the stopping conditions been met.

update decides whether or not *p* should get added to *p_set*, and perhaps, if it is added, which other element of *p_set* to delete. This could again be a function of the iteration number.

reorganize analyzes the set of partitions and makes adjustments to the set. The idea is to look over the results so far and try to focus the search in high payoff areas.

When the decision to keep a partition for later analysis depends only on that partition, and not on what the other partitions look like, a second form of *find_partition* is possible that does not keep *p_set* in memory. Instead, depending on the FOM being good enough, it goes ahead and computes higher and higher quality FOMs. The code for this might be as in Figure 3. *is_acceptable* is an evaluation of the partition that generally gets more accurate (and slower) for increasing *i*.

The AFD program of PT 1 is essentially of this second form. AFD Version 1 goes through all non-shared variable partitions exhaustively. It has two iterations. The first uses the existence of features for *is_acceptable*. The second iteration uses DFC of the children (as computed recursively by this version). AFD Version 2 is the same as 1 except it uses a slightly higher threshold on ν, i.e. the simple decomposition must be negative. AFD Version 2b is the same as 2 except the criteria for a negative simple decomposition is based on DPFC rather than DFC. AFD Version 2a is the same as 2 except the variable partitions are not exhausted. Instead, the partitions

```
while (not_done(0)){
        p = choose_partition();
        i = 0;
        while ( p->is_acceptable(i) && i < no_of_iterations ) i++;
        if (p is_better_than best_so_far) best_so_far = p;
};
```

Figure 3: A second find partition function.

are considered in order of increasing number of column variables until a negative decomposition is found. AFD Versions 3 and 4 are the same as 1 and 2 respectively, except shared variable partitions are included.

7 Conclusion

This memo discusses the options for function decomposition that will be implemented in the Function Learning and Synthesis Hotbed (FLASH). FLASH was designed to allow easy selection between these options so their speed and effectiveness could be evaluated. Future memos will describe the results of such evaluations.

References

[1] Mike Breen. *Some Results in Machine-Learning.* Final Report, USAF Summer Faculty Research Program, 1992.

[2] Micheal J. Noviskey. *A Correlation Strategy for Construction Promising Partitions in Decomposing Functions.* Technical Memorandum AFWAL-TM- (TBD), Wright Laboratory, June 1992. (in preparation).

[3] Timothy D. Ross, Michael J. Noviskey, Timothy N. Taylor, and David A. Gadd. *Pattern Theory: An Engineering Paradigm for Algorithm Design.* Final Technical Report WL-TR-91-1060, Wright Laboratory, USAF, WL/AART, WPAFB, OH 45433-6543, August 1991.

[4] James S. Wolper. *Aspects of Pattern Theory.* Final Report, USAF-RDL Summer Faculty Research Program, August 1991.